Revising the worksheet with L3: a language and environment for user-script interaction

Michael Hohn[1,*,†]

[1] *Lawrence Berkeley National Laboratory, One Cyclotron Road, MS 64R0121, Berkeley, CA 94720, USA*

SUMMARY

This paper describes a novel approach to the parameter and data handling issues commonly found in experimental scientific computing and scripting in general. The approach is based on the familiar combination of scripting language and user interface, but using a language expressly designed for user interaction and convenience. The L3 language combines programming facilities of procedural and functional languages with the persistence and need-based evaluation of data flow languages. It is implemented in Python, has access to all Python libraries, and retains almost complete source code compatibility to allow simple movement of code between the languages. The worksheet interface uses metadata produced by L3 to provide selection of values through the script itself and allow users to dynamically evolve scripts without re-running the prior versions. Scripts can be edited via text editors or manipulated as structures on a drawing canvas. Computed values are valid scripts and can be used further in other scripts via simple copy-and-paste operations. The implementation is freely available under an open-source license.

key words: scientific computing; scripting languages; human-computer interaction; graphical user interfaces; end-user programming; application development

[*]Correspondence to: Michael Hohn, Lawrence Berkeley National Laboratory, One Cyclotron Road, MS 64R0121, Berkeley, CA 94720, USA
[†]E-mail: mhhohn@lbl.gov
Contract/grant sponsor: NIH/NIGMS; contract/grant number: P01GM064692

INTRODUCTION

The use of two languages to split an application into a low-level kernel and a high-level scripting language is now quite common and has given rise to a large number of general extension languages, including Tcl [1], Lua [2], and Python [3], as well as languages specialized for numerical computing such as MATLAB [4], Octave [5], R [6], and others. The appeal of this approach is its flexibility: users are not restricted to using files to provide a large number of parameters to monolithic programs, but are free to assemble customized programs from smaller modules. Aside from speed and space issues, the technical distinctions between the scripting language and the kernel language are of little interest to users. The distinction that matters most is convenience: scripting languages have no user-visible compilation steps, require no type declarations, and integrate topic-specific high-level facilities in the language. The scripting languages are well-suited as data input languages, avoiding the need for a custom data file format. In short, scripting languages perform many duties for the user, at the cost of performance and increased memory use.

These high-level languages are typically used in two ways, through an interactive session, or as stand-alone programs. Used as standalone programs, parameters may be embedded in the script or read from files, the program is executed, and the computed values are saved to disk for further processing. The interactive sessions give full access to the language and provide data access via variables in the topmost workspace; when done, the workspace can be saved and later revisited (in MATLAB and R).

Unless the problems solved require only a fixed sequence of steps, both cases introduce an (edit / run / examine) cycle. In this cycle parameters and computed values change, and often collection of values needs to be retained for later examination. Using an interactive session for this cycle is ideal: there are no data files to name, read or write; scripts (or parts of one) are simply re-run as needed, and the scripting language often provides sufficient data structuring facilities to keep such sessions well organized. All of these languages offer some data structuring facilities within the language, and several provide full interactive environments including facilities to inspect values in the topmost workspace. When a worksheet interface is available, interactive sessions can often be re-run from a selected point in a toplevel script. This provides a simple local means of adjusting parameters until satisfactory results are obtained. When a collection of values needs to be retained for comparison, scripts can be adjusted to accumulate them in appropriate structures and rerun. Unfortunately single sessions are not sufficient for solving real problems for several reasons. First, many computational steps take a long time to complete and can therefore not be run repeatedly; they are run once and the data is saved. Second, no single language or tool provides all facilities needed in experimental computing so often several data sets need to be kept for later work using other tools or scripts.

The need for persistent data moves the above cycle out of the scripting language's environment, to the level of the interactive shell and into the file system. The language's

structuring facilities are lost when writing data out for long-term storage. While the interaction cycle from a shell is the same (editing / run / examine), the resulting data accumulation and organization can be daunting; the naming, grouping, and examining of a large number of parameters, scripts, and data files are all done *manually*.

To alleviate the burden of manually tracking files and parameters and to give a higher-level overview of a computation, data-flow approaches are commonly used. In addition to the venerable `make` [7], several visual environments have been developed, including SCIRun [8] and Viper [9]. In operation, they are similar to `make`: one constructs a program as directed acyclic data flow graph, with nodes holding procedural language code for actions, and the system evaluates nodes in topological order. All three provide persistence by storing data on disk; because every node corresponds to one datum, data selection is simple and the datum's context is explicit. Data can be examined at any time. They also provide need-based evaluation; re-running a data flow graph visits every node, but only executes new nodes or nodes with changed dependencies. This makes additions simple and avoids redundant effort when new nodes are added.

On the other hand, the data flow approach lacks general programming facilities, nested function calls and looping constructs in particular. While looping within a node is possible, repeating a whole subgraph with slightly different parameters (and examining new values) requires cloning the whole graph. Splitting a long script or a script containing data to be examined into nodes, or combining nodes into a single script are not simple operations. Thus, switching between a data flow system and a scripting language can be difficult.

In summary, the advantages of direct scripting are conciseness, flexibility, and familiarity, while data flow provides clear data organization, has a clear data-to-script association, is implicitly persistent, and provides need-based evaluation.

This paper describes L3, a language and interface designed for scripting and interacting with data using a single unified interface. L3 combines features from the procedural, functional, and data flow approaches to application control in a single language to make it suitable for both the general data handling problems encountered in script use (primarily numerical computing) and as infrastructure for a persistent graphical data handling interface.

Just as two-language systems use the higher-level language to "glue" together applications, the use of L3 "glues" together the applications and the data they produce, providing a simplified interaction between user and script.

Integrating data flow and procedural facilities in one language removes the separation (and resulting discrepancies) between a high-level graphical workflow (often a diagram) and the more detailed script implementation. By design, L3 scripts correspond directly to a nested data flow graph, and this graph can be automatically produced from l3 scripts.

L3 is implemented in Python and shares (a subset of) its syntax. L3 itself provides no constructs for large-scale programming, such as classes, iterators and modules. Instead, L3 is extended through Python, just as Python is extended though C or C++. Unlike a Python/C connection the L3/Python connection requires no special code, so migration

between scripts and workflow amounts to moving text between files. If a Python script contains data to be examined, it can be moved “up” to L3; conversely, once a L3 script has stabilized, it can be moved “down” to Python.

AN OVERVIEW OF THE LANGUAGE

L3 is a special-purpose functional scripting language that offers a one-to-one bidirectional connection between programs and computed data. Through need-based evaluation, it allows incremental development of scripts, i.e., scripts may be written, interpreted, extended, and interpreted again, and only the additions will be interpreted. All data produced during execution is persistent and may be examined using the graphical interface. The syntax is described first, followed by the novel features.

Syntax

The L3 language syntax closely follows Python’s for compatibility, using reserved words and indentation to delimit block structure.

The basic types are `int`, `float` and `string`. Data structures provided by L3 are (nested) lists (`["a", 1, "b", 2.2]`), tuples (`(1,2)`), and dictionaries (`{ a = 1, b = 2.2 }`).

Expression syntax and precedence follows Python’s and includes the comma-separated function call syntax (`add(a, b)`), the usual arithmetic operators (`-`, `+`, `%`, `*`, `/`, `**`), the comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) and boolean operations `and` and `or`. Also included are the array slice operators commonly found in numerical scripting, e.g. `a[1:10]` to retrieve entries 1 through 9 of `a`.

Programs are structured via conditionals, functions, and dictionaries. Dictionaries in L3 are nested environments containing a sequence of expressions; evaluation returns a key-value mapping. In the simplest use with comma-separated assignments, `{ a=1, b=2, ...}`, this reduces to the expected dictionary behavior. Scoping for dictionaries is lexical, so scripts can be nested without naming conflicts. Functions can return multiple values, and tuple assignments can be used to bind their components. Functions are tail recursive and lexically scoped, so complex loops can be written as tail calls. For simpler loops and compatibility with Python, the `for` and `while` loops are also provided.

Other Python types and classes are treated as opaque objects, are constructed via function calls, and accessed through their member functions (`list.append(1)`).

The L3/Python connection is provided via the `inline` special form; it evaluates Python statements and makes bound names available to L3. Using this approach, arrays and other subject-specific data structures are provided through Python and accessed via operator overloading or explicit function calls.

Some of these features are shown in figure 1. This short example uses L3 built-in types, as well as two Python extension libraries `numpy` and `sparx`. The `volume` is a three-

```
1  #* Set up environment.
2  inline "import sparx as sx; import l3lang.external.sparx_mixins"
3  inline "import numpy as N"
4
5  #* Get starting volume.
6  volume = sx.getImage( "model.tcp" )
7
8  #* Form raw projection.
9  raw = sx.project(volume, [10.0, 20.0, 30.0, 0.0, 0.0], 35)
10
11 #* Form noise.
12 noise = sx.model_gauss_noise(0.4, raw.nx, raw.ny)
13
14 #* Form projection + noise.
15 range_n = noise.maximum - noise.minimum
16 range_raw = raw.maximum - raw.minimum
17
18 def noisy_image(nsr):
19     return nsr * range_raw / range_n * (noise - noise.minimum) + raw
20
21 # Produce several noisy images for inspection.
22 for nsr in N.arange(0.0, 3.1, 0.1):
23     noisy_image(nsr)
```

Figure 1. Example of a L3 script using Python as embedded language.

dimensional array type implemented externally, while `noise` and `raw` are external two-dimensional arrays. `range_n` and `range_raw` are defined in the same lexical environment as `noisy_image`, making them available in the function body (line 19); they are scalar and Python's operator overloading is used in `noisy_image` to provide standard expression syntax for the mixed expression. The for-loop (line 22) only forms the noisy images, but leaves their retention to L3.

Except for the `inline` statement, this script is fully compatible with Python. If it is run without errors, the results are identical.

Persistence

The script of figure 1 can be run interactively, while a long-running script can be run in batch mode. Both cases produce one additional state file containing *all* values calculated in the session or script, including intermediate expressions and function bodies. When this state is openend along with the script the toplevel behaves as if the script had been run in it.

This heavy use of memory and disk space brings the convenience and other benefits of a worksheet environment to long-running scripts. While a fast calculation like figure 1 could be done repeatedly within a single environment and would not require persistence, long-running scripts require storage of data for later examination of results.

Nesting

Similar to data flow, L3 persistence keeps not only programs and data, but also the information connecting them. Persistence in L3 extends to function calls and loops so every value formed during execution is kept, not only those at the top level.

After the script of figure 1 is run, all values of `return nsr * ...` (line 19) can be retrieved and inspected. Simply itemizing these in a list has limited use, as one needs to know what distinguishes individual results. Thus, for any one of these values, L3 can identify the corresponding `nsr` value.

Further, L3 links computed data to its source expression and the expression to the environment in effect at the time it was evaluated. This way, the multiple values produced by expressions in loops and functions can all be inspected in the context that produced them. So, if the function body had other computed values, one set for every `nsr`, they could be inspected for a chosen `nsr` as well.

Context is not limited to function arguments. For example, running the code

```
for x in range(0,4):
    E = [x, x**2]
```

results in the values

```
[0, 0.0]
[1, 1.0]
[2, 4.0]
[3, 9.0]
```

The backlink information for one of these, say `[2, 4.0]`, identifies not only `[x, x**2]` but also the matching repetition of the `for` loop, showing the value of `x` and any other variable defined in that iteration.

This ability to inspect data computed inside a loop or function is very important in experimenting with scripts. The script currently being used is often pushed into a function of its own, but the values it produces still need to be examined in context[†].

There are many useful tools external to Python and L3, and many of these use their own file and data formats, and expect a certain file organization. Thus, the computed

[†]Computed values could of course be returned explicitly, and the aggregate could then be examined. But this requires more code (to form the aggregate), makes examination more difficult (traversing the aggregate), and loses the context of the computed value.

data may have to be written out explicitly after all. In this case, L3 can provide unique file names for the data; computed results are written to disk as before, but L3 keeps the script/data links and other meta-information. Using one indirection for the file name (`myfile = new_id()` instead of `"myfile"`) seamlessly integrates files into L3, resulting in lexically scoped treatment of external files. Instead of using

```
compute_and_save(param = 1, "myfile")
```

and adjusting for a loop via loop-dependent file naming

```
for ii in range(0, 11):
    compute_and_save(param = ii, "myfile_%d" % ii)
```

and manually handling the resulting files, one can simply use

```
datafile = "myfile-%d" % new_id()
compute_and_save(param = 1, datafile)
```

Moving this to a loop requires no file handling change, because loop bodies are unique:

```
for ii in range(0, 11):
    datafile = "myfile-%d" % new_id()
    compute_and_save(param = ii, datafile)
```

This is again illustrated in a later example.

Need-based evaluation

In data flow environments, adding new nodes (operators) that use existing data is a common operation. It is also cheap, because only the new nodes are executed. L3 offers the same feature: expressions can be inserted in lists and dictionaries at any time. When the containing program is re-run, only new expressions are evaluated.

The implicit persistence keeps these scripts simple by avoiding extra code for input / output and file naming. The addition of need-based evaluation allows scripts to evolve without re-running everything. By placing new code in existing scripts, the environment (parameters, file names, loop variables and function call arguments) already provided by the scripts need not be replicated.

Unlike a flat workspace environment, the new expression can be added anywhere in a script. In particular, it can be added to a deeply nested function body or loop, and it will evaluate for every call or iteration as if it had been there from the beginning. This avoids explicit file I/O, re-writing of surrounding iteration code, and repetition of long-running operations. The loop

```
for ii in range(0,101):
    x = ii/100 * pi
    [x, sin(x)]
```

is trivial and if an addition is made, the whole loop is simply re-run. But when we have something of the form

```
for ii in [COMPLEX_ITERATION]:
    EXPENSIVE_FUNCTION(ii)
```

re-running the whole loop for an addition is no longer practical, and recreating the iteration for the addition is equally undesirable. This is illustrated in a later example.

Error handling

Errors are assumed to require manual intervention, so L3 provides no exception handling facilities. Instead, exceptions from the host language (Python) are treated as stop instruction. In interactive sessions, control is returned to the user; in batch mode, the current state is saved, and the interpreter exits. In both cases restarted execution continues from the last successful evaluation, allowing the user to fix the erroneous expression and continue from there.

OVERVIEW OF THE INTERFACE

Graphical script representation

L3 serves as scripting language and as data examination tool, so the interface is a hybrid of a pure text representation using scripts and a structural representation needed for manipulation. Interaction with data is simplest via a point-and-click interface, while writing text is the established way of producing scripts. In the literature and from experience, we find that graphical construction of complex and detailed code via graphs is impractical [10], [11], but simple or subject-specific constructs can be meaningfully represented graphically [12] [9] [13], [11].

The L3 interface takes a mixed approach. The textual sample of figure 1 is displayed as shown in figure 2. Lists, functions, conditionals, outlines and loops have explicit structures resembling the text layout but include markers, the grey bars between successive script lines. All other expressions are shown in textual form. Comments in scripts are visually attached to the following expression, but are not part of the script proper. They can be changed without affecting L3 execution in any way; in particular, they will not cause re-running of an expression.

To provide a simple overview of larger scripts, L3 uses a collapsing outline structure as commonly found in other tools. In L3 the outline is part of the script itself, embedded via `if` as shown in figure 3. This way, the structural editing operations in the GUI are well-defined, and the L3 script remains valid in Python. As may be expected of markup, the input file's readability decreases, while the rendered output (figure 4) improves.

▾A. Program

```
Set up environment.
inline "import sparx as sx; import l3lang.external.sparx_mixins"
inline "import numpy as N"
Get starting volume.
volume = sx.getImage( "model.tcp" )
Form raw projection.
raw = sx.project(volume, [10.0, 20.0, 30.0, 0.0, 0.0], 35)
Form noise.
noise = sx.model_gauss_noise(0.4, raw.nx, raw.ny)
Form projection + noise.
range_n = noise.maximum - noise.minimum
range_raw = raw.maximum - raw.minimum

noisy_image = Function
    nsr
    return nsr * range_raw / range_n * (noise - noise.minimum) + raw

Produce several noisy images for inspection.
for nsr in N.arange(0.0, 3.1, 0.1)
    noisy_image(nsr)
```

Figure 2. Displayed version of the textual script of figure 1.

The worksheet model

The l3 language provides a foundation for interactive interfaces but fundamentally only requires a program and an environment to execute it in. When executing a script from the command line, there is one environment per program. In a worksheet environment, sharing of named data between scripts is very convenient, so the graphical interface uses one environment per worksheet shared among all programs on that worksheet.

Shown in figure 5 is the startup screen. On the right side, the worksheet interface is modeled as a single sheet of paper, with one name binding environment. On the sheet are one or more scripts, sharing this environment. A second sheet on the left, with its own environment, provides short introductions, examples, and a library of available language constructs. For specific applications, this library can be supplemented with templates to be adjusted by users.

```
#* A self-contained program for image processing.  From a
# volume, images are projected in known directions with known
# angles;  an alignment algorithm is called to find the number
# of directions, and reverse the angular spread.
program:
    if ("outline",  "SPARX volume projection and image alignment"):
        #* Form the test data, projections from volume.
        if ("outline", "Project volume."):
            if ("outline", "Preparation."):
                inline('''if 1:
                       from EMAN2  import *
                       from sparx  import *
                       from random import randint
                       import l3lang.external.sparx_mixins
                       import os
                       ''')
                ev = os.path.expandvars

            if ("outline", "Load volume."):
                vol = getImage(ev("$SPXROOT/sparx/test/model001.tcp"))
```

Figure 3. A script incorporating an outline via the `if` construct. The full script's outline is shown in figure 4.

Saving the gui state records the state of both sheets in a single file, so that multiple scripts can share data, and additions can be made to the library as useful scripts are developed.

Everything is interactive

In the L3 language, everything is an expression. In the display, every expression is an interactive graphical object that can be manipulated directly or through its own menu. Direct manipulations of expressions, performed via mouse clicks and drags, include selection, copy, move, attach and detach. All other operations for expressions are available via menus and include deletion, inspection, conversion to text, textual editing, and display adjustment of the expression itself; they also include printing and insertion of value(s).

Script construction and modification

The dual text / object representation requires multiple ways of constructing and interacting with a L3 script. Writing code is the primary approach for initial program construction. This is familiar to programmers and exposes every detail of a script. There are three ways to insert scripts in a worksheet. For scripts in files, the whole file can be inserted. Sample code from other text sources (web pages, emails, etc.) can be directly

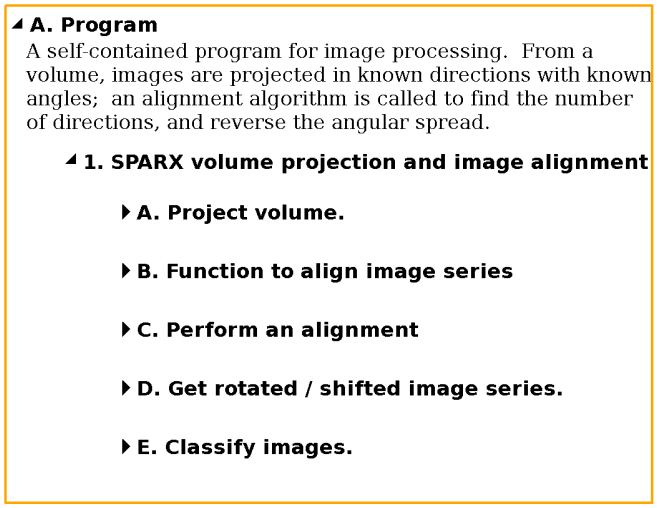

Figure 4. A script from the library (figure 5), viewed as an outline. The outlines can be collapsed and expanded via menus, or by clicking the triangles. As everything in L3, outlines are expressions and can be detached, moved, and attached elsewhere while retaining their associated data. This program is a copy of entry 1.B.1 from figure 5, with source representation shown in figure 3.

pasted on the canvas. And for immediate feedback a toplevel can be coupled to a program on the canvas; every expression entered is executed and appended to the program.

Once in the worksheet, expressions can be handled as objects and new ones constructed from existing parts. This approach is useful when starting a new project using existing scripts, as it provides a cleaner copy-and-paste approach than raw text. It can be especially useful when providing tools for beginners, by giving only a few selected templates with limited plug-in choices. The availability of insertion markers for expressions could be used to construct whole programs from scratch, but it is not practical for that — for low-level scripts, it would be far more tedious than editing text.

Handling expressions as objects is also the only way to add expressions to a program that has already executed. Adding new expressions to scripts using insertion markers makes the change local to the chosen expression, and all surrounding expressions retain links to the data previously produced. This feature allows mixing of data examination and script execution, and simplifies the (run / edit / examine) cycle.

One example of this is a loop in which every iteration takes a long time, requires prior iterations' output, and may not lead to good results. So we run the loop

```
while error(result) < high_tolerance
    result = expensive_function(result)
```

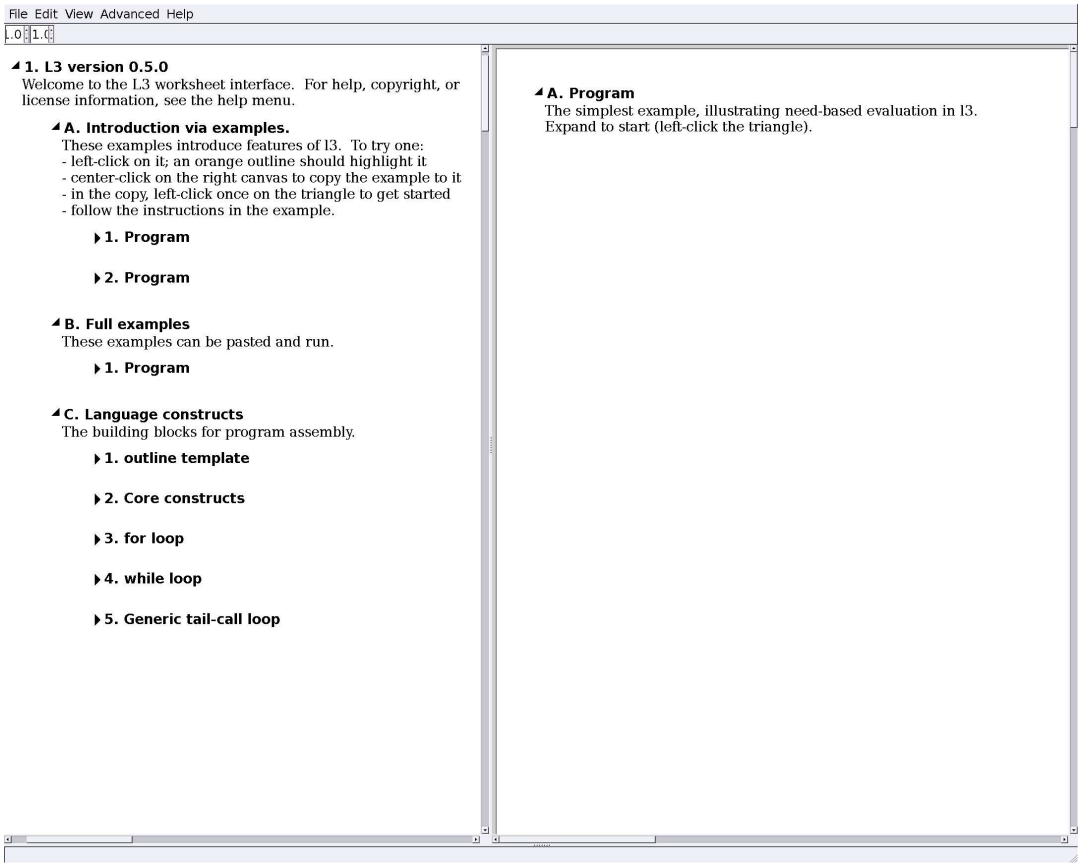

Figure 5. The L3 project interface.

and examine the intermediate `results`. If they are promising, we change `high_tolerance` to `low_tolerance` via structural editing, and run the script again, automatically reusing all previously computed data. More common uses of this insertion ability are illustrated later.

Data examination and reuse

There are two ways of handling computed values in L3. As usual in scripting languages, values bound to names can be used in other parts of the script through the name[‡], and values are passed to functions by reference or to other scripts via reference to a file.

The alternative way of using values in other scripts is via direct selection and insertion on the canvas; this is equivalent to explicitly giving the datum a global name and referring to it, but explicit naming of the value is avoided. No intermediate textual copy is formed in the worksheet, and every value retains a link to its creating expression. For example, after a script containing `10 - 2` is executed, the expression's value `8` can be copied to the worksheet and pasted into another expression.

There is no distinction in the handling of a value and an expression – L3 data is always a valid L3 expression. Computed data in L3 is either a native Python type (by extension, any application-defined type) and is treated as simple value[§], or it is one of the recognized nested types, tuple, list, or dictionary. Therefore L3 can serve as editor for data with nested structure. The simplest example is a list of values, in which values can be added or removed, or the whole list can be inserted elsewhere. Also common are lists of tuples, which are displayed in a tabular form.

AN EXAMPLE SESSION

This section illustrates how the *combination* of the individual features of L3 can significantly reduce the manual effort required when using scripts and programs experimentally. A comparison is done by describing the same computing session as done via L3, within a scripting language's environment, and using a script from the shell. Although these examples use Python, the concepts apply unchanged to other scripting languages.

The whole script is an experiment to test the algorithm described in [14], and to identify best choices for certain parameters. The only difference between this test script and an actual user session is the source and quantity of the data. As is typical in experimental interaction, there will be more than one run once some insight has been gained. If subsequent trials yield more insight, more modifications may follow, etc. Eventually, these script modifications must be cleaned up and the program / file structure must be redesigned, but that point is reached much later when using L3, providing better focus on the problem at hand.

[‡] Name lookup is lexical, so a name reference is only valid in the same scope, or a nested one.
[§] Simple values receive no special handling by L3 unless special handlers have been added, e.g. for images. Also, values in the worksheet interface need not have a parseable representation as long as they can be converted to a binary string.

Direct interaction

The example starts in the middle of a data analysis sesssion. Some values computed by the script in figure 4 need to be examined manually to find the best choice of data set for a subsequent computation.

The L3 approach

Section E of the sample script from figure 4, fully expanded for inspection, is shown in the top part of figure 6. To choose the best result, we need to examine the values produced by the call of `k_means_info`. From `k_means_info`'s menu at ❶, choosing "insert values" results in the insertion at ❷. Every value is a tuple, and the function is called inside a loop, so "insert values" returned a list of tuples and displays it.

From the script (①) we see that the the parameters `coleman`, `harabasz`, ... form the data columns. Because the values are formed in the loop over `K`, we know the values have corresponding `K`'s, but are shown without them; this will be remedied later. For now, the `K` value is seen in the third column (③) as part of the file name.

With the data in a simple table, we can now look for the row, if any, with the smallest value of `coleman` and the largest value of `harabasz`. Such a row doesn't exist, but row ❸ comes close. This row contains the selection parameters and names of files holding data to be inspected. Here we want to view the averaged images, found in the file at ③. The session continues by

- selecting the file name's "insert value" menu (③),
- collapsing the function call (❶),
- removing the table row of interest (where `K` = 8, ❸),
- and collapsing the value selection (❷),

resulting in the session shown in figure 7. The file content is again displayed as L3 data (❹), in this case a list of (index, image) pairs.

This simple example shows several practical results of L3 features: user efficiency, clarity and flexibility. The entire interaction is is done in less time than it takes to read this description. By compacting, not erasing, the already examined data, we retain a full overview. We can see the script's outline, the current section's details (the context), the function whose values were examined, an expandable box holding those values, the relevant row from those values, and the contents of a file from that row.

The interaction also demonstrates the utility of expression nesting and of unifying expressions and values. Starting from the script, we get a list of tuples of values; choosing one of those values (the file name), we again get a list of tuples (the index, image pairs in the file). This nesting can continue arbitrarily, imposing no data structuring limitations. Every value displayed is again a valid expression, and could be copied (by reference) to another program.

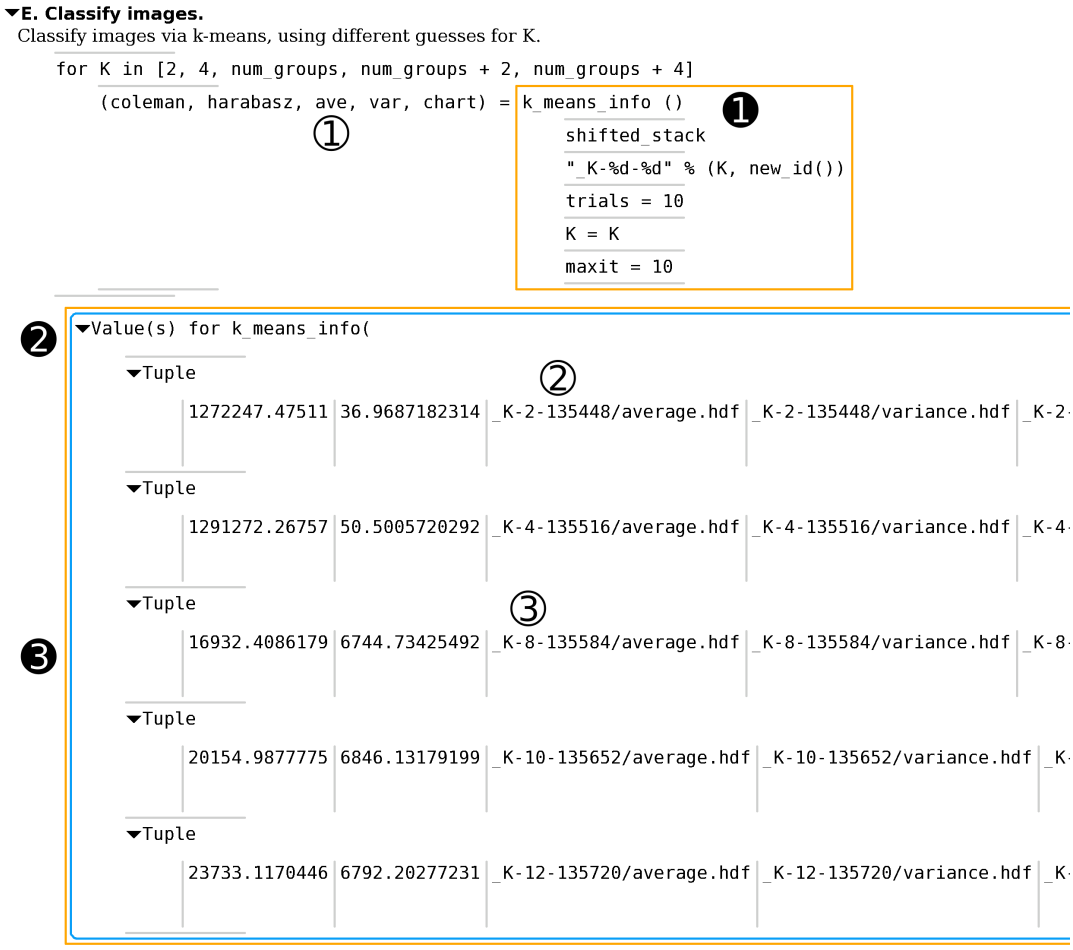

Figure 6. Selecting multiple values accumulated inside a loop via a menu selection. This script is part E. from figure 4. Every value is a tuple, so a list of tuples is inserted. For these values, the assignment names (coleman, harabasz, ave, var, chart) serve as column headers.

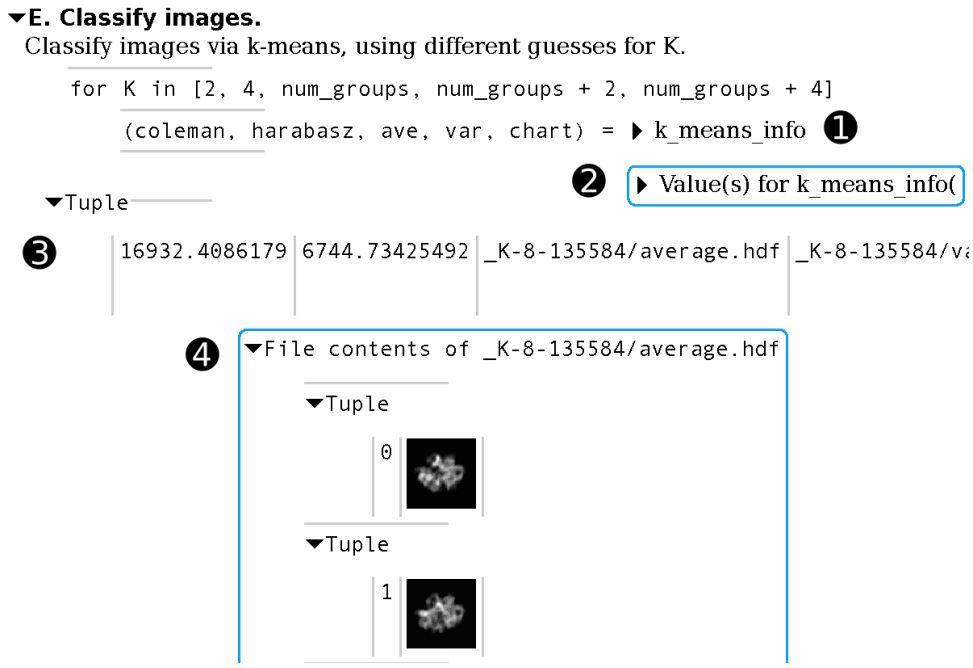

Figure 7. Accessing a file's content via the file name. Extracting row ❸ from figure 6, and hiding ❶ and ❷, only the row ❸remains. In that row, the the file _K-8-135584/average is chosen and displayed. It contains a list of images, so the display (❹) shows (index, image) tuples.

The whole test script is long and has many parts that could be examined similarly, but for comparison of effort we next examine the preceding steps as done from the shell level.

A shell-level approach

The script itself only computes values, but contains no code for accumulation of data. For this long-running script, keeping the information in a separate file using a simple log format is a reasonable approach[¶]; because similar log entries can be expected in other

[¶]The data cannot be returned as a value because it is computed deeply inside the script, and there are other data that would need to be returned as well. The data has to be accumulated as a side effect, and because this script is run in batch mode this accumulation must be saved to a file.

A more thorough approach could accumulate real data structures in a separate file, but this is significantly more effort than simple text printing and for these "terminal" values there is no advantage.

```
1 if ('outline', 'Classify images.'):
2     print "Image classification result: (coleman, harabasz, ave, var, chart)"
3     for K in [2, 4, num_groups, num_groups + 2, num_groups + 4]:
4         (coleman, harabasz, ave, var, chart) = k_means_info(
5             shifted_stack, "_K-%d-%d" % (K, new_id()),
6             trials = 10, K = K, maxit = 10)
7         print "Image classification result:", \
8               (coleman, harabasz, ave, var, chart)
```

Figure 8. Section E of the script with logging output added in lines 2 and 7.

```
Image classification result: (1272247.47, 36.9687, '_K-2-135448/average.hdf', ...
Image classification result: (1291272.26, 50.5005, '_K-4-135516/average.hdf', ...
Image classification result: (16932.4086, 6744.73, '_K-8-135584/average.hdf', ...
Image classification result: (20154.9877, 6846.13, '_K-10-135652/average.hdf',...
Image classification result: (23733.1170, 6792.20, '_K-12-135720/average.hdf',...
```

Figure 9. Log fragment produced by script in figure 8, truncated to fit.

parts of the script (and other scripts if writing to one log), the minimal additions are lines 2 and 7–8 in figure 8. This of course must be added before the script is run, which was not done above, leaving two choices. Either we change the original script (the change is only in section E), re-run it and incur a time penalty for recomputing *all* data (sections A-E), or we separate section E of the script from the rest and introduce another script, hoping to avoid the recomputing step. In the latter case, we have to add code for saving and loading the data to the new scripts. Further, the `num_groups` parameter also must be passed between the scripts in some way. The added data saving will require re-running the modified first script, removing any time savings.

We choose to change the original script and re-run it. Once done, we must look for the relevant information in a log file which contains significantly more than this. Looking at the relevant lines in the log (figure 9), we can choose the best result as before, the line

```
Image classification result ... '_K-8-135584/average.hdf', ...
```

and use an external viewer to inspect the file `_K-8-135584/average.hdf`.

Although this session only serves to examine some data, it got complicated. All together we have three disjoint windows — script, log, and viewer — for the information found in the script, the log file, and other files. The script got more complicated because it not only computes data, but explicitly retains it. Any extra information, such as intermediate file names or the steps taken to reach the selected data file, must be written down by the user.

Data extraction via expressions

The previous sample session illustrated the convenience of direct access to data. The combination of persistence and need-based evaluation extend this to situations where additional work is performed on nested data. To illustrate this, we continue the session as follows. We are experimenting, and decide that graphing the criteria `coleman`, `harabasz` against `K` would make sense. Assuming a graphing utility is handy, how do we get the data to it?

Via L3

Taking advantage of the incremental evaluation in L3, we simply embed the new expressions `(K, coleman)` and `(K, harabasz)` into the original L3 script so it becomes

```
if ('outline', 'Classify images.'):
    for K in [2, 4, num_groups, num_groups + 2, num_groups + 4]:
        (coleman, harabasz, ave, var, chart) = k_means_info(
            shifted_stack, "_K-%d-%d" % (K, new_id()),
            trials = 10, K = K, maxit = 10)
        (K, coleman)
        (K, harabasz)
```

Re-running the whole script only executes the new expressions, a list of `(K, coleman)` values is retrieved via a single mouse selection, and the `plot` function is applied to this list.

This approach scales to much more complex scenarios — nested function calls, nested loops — because the most complex parts (establishing the context for the iteration, retrieving the input data, writing the output data) are all implicitly handled in L3.

Via scripting

This loop is part of a larger script, so we again opt for the simpler approach of changing and re-running the script. Inserting appropriate code to form a list of (K, value) pairs results in code like

```
K_cole = []
K_hara = []
for K in [2, 4, num_groups, num_groups + 2, num_groups + 4]:
    (coleman, harabasz, ave, var, chart) = k_means_info(
        shifted_stack, "_K-%d-%d" % (K, new_id()),
        trials = 10, K = K, maxit = 10)
    K_cole.append( (K, coleman) )
    K_hara.append( (K, harabasz) )
```

In the interactive session, but separate from the script, we can use variations of

```
plot(K_cole)
plot(K_hara)
```

to examine the data.

While the change is straightforward, re-running the script is expensive. The simple test case considered here takes several minutes, while a real run could take hours. Also, this approach is for single interactive sessions only. For a long-running program, explicit file I/O would be introduced.

Via the shell

The log file (figure 9) is only marginally structured; luckily, all information is included in the log, and on independent single lines; we need only extract columns 1 and 2 and the `K` value. Using the unix tool approach, we write a short script using regular expressions to extract the data from the log and graph it.

There are several shortcomings to this. Log files are likely to contain many more "Image classification result" lines, and those would have to be further distinguished to ensure the correct set is extracted. Changing the log format in any way requires fixing the script, while further exploration of data may require writing more information to the log files. Very soon, log files may have to be parsed in more careful ways, making it difficult to pass data on for other use.

The fundamental problem here is that useful data is treated as a side effect by writing it out to a log instead of keeping it with other computed data as done by the scripting approach. While this keeps the original scripts simpler, the resulting data handling is much more complicated.

Manual repetition

The two previous sample sessions illustrated the convenience of direct access to data combined with persistence and need-based evaluation. The following session describes a simple method for manual repetition of program fragments, using lexical scoping to avoid name clashes, allowing a cut-and-paste style of interaction to work. After the preceeding examination, we want to try some other parameters in the script, rerun it, and perform a similar examination. To see whether our new version is better, we also want to compare results from the previous session with those of the new one.

The L3 approach

In L3, the user effort for this is one copy of a `dict` (to get a separate namespace), followed by operations on the script: one selection, one copy, modification of parameters, and a new run (roughly five button clicks, plus the parameter modification), followed by whatever

inspection is desired, as described above. Because all computed data is available as data structure, pairwise comparisons are simple. Although most data read and written in the script resides in files, there will be no naming conflicts when using L3's `new_id`. The combination of lexical scoping and function cloning in L3 avoids clashes between multiple uses of a single script, and `new_id` extends this to external entities.

The shell approach

In the shell or a script, more care is required. Naming conflicts must be avoided during the creation of the new data. A new set of parameters requires one new file, in this case a copy of the script. The parameters can be adjusted in this new file. The data read by the new script is the same as before, but the output data is new and different from the previous run and must be distinguished. For this, the file names must also be adjusted[‖]. Once this new script has run, we again need to identify the best data set. To do this correctly given the now present collection of files and procedures, we can *manually* follow the description of the log mechanism previously described, or other notes taken.

Programmed repetition

This last session describes a simple method for automatic repetition followed by manual examination of produced data.

The previous parameter changes show promise. To improve results further, we want to try the script on a collection of new parameters and look at the results sometime (much) later.

The l3 approach

Taking advantage of l3's lexical scoping, we move the whole script into a function, changing the embedded parameters to function arguments, and call the new function as needed. Because l3 expressions are globally unique, any script can serve as the body of a function, even those writing to files (as long as the file name incorporates `new_id`). We get one more script copy for a total of three explicit versions: the original, the manual test, and the newly formed function. Remaining tests are done via function calls. Once the new function has been run, values are chosen as before, through the function body. Because there are multiple calls to the function, the call of interest can be used to narrow the selection.

[‖] If the script is run from the same directory, the input file names remain unchanged, but the output file names must be adjusted. If a new directory is created for the new output, the input file names need adjustment.

The script approach

This change is also valid in a scripting language, but values computed inside nested functions are not visible at the top level. Simply computing the extra values is not enough; they also need to be made available at the toplevel. As for the previous incremental change this requires accumulating them, either via files or data structures. Following this, the resulting structures of files need to be retrieved and examined explicitly.

The shell approach

This degree of change breaks the shell approach completely. To accomodate multiple calls, several options exists; none are clean or simple. One way is to add the new parameters to the script's arguments list. This either requires adding argument parsing if not present, or extending the argument list. If the argument list is extended a few times, the script will literally have dozens of parameters, most of which serve only to control side effects and distract from the main purpose of the script. Another way is to introduce a new script that generates parameter files or scripts and then executes them. This is of course macro-expansion done at the shell level, and incurs all the well-known problems thereof.

DISCUSSION

Use as a debugger

L3 shares many properties with traditional debuggers. The ability to inspect all data after execution, including those produced inside function calls and loops, can be used in the development of algorithms prototypes. Examining intermediate results for consistency with expected behavior, other models, etc. is a common step in algorithm implementation.

Portability

The language itself runs wherever (the C version of) Python is available and is therefore very portable. For long-running jobs on computing clusters, l3 scripts can be run in batch mode, and the results later examined through the graphical interface on a desktop machine. The worksheet interface is as portable as the toolkit used; currently gtk [15] runs on all major desktop systems with graphical interface.

Compatibility with existing code

The use of Python syntax where possible avoids the need to learn a new language, and embedding Python provides trivial access to a broad library. But there are some differences restricting the functions and types that can be used from L3. Full use of

L3 requires persistence of all Python values returned to L3. This precludes the use of generators, file descriptors, and sockets in L3 scripts. This is not a problem in practice as all of these can be used within Python functions called from L3, as long as such access is atomic to L3. For single sessions using the worksheet, the persistence requirement can be ignored.

Embedding other scripting languages

The L3 language is very small and simple and represents a subset common to Python, Scheme, Matlab and Maple; although there are clear differences between the capabilities of these different languages, L3 could be implemented on top of them as easily as Python. The L3 worksheet could not be ported so easily, but by fully separating it from the language backend, it could serve as interface to a different language.

Use with external programs

The L3 language is designed as layer on top of Python and automates data handling for Python-provided functions. By controlling external programs from L3 or Python, all data files, parameters, program calls, and output is again tracked and accessible from a single worksheet containing one or more scripts.

Two approaches have been tried to accomodate external programs. The first is to write a Python wrapper for each program; the wrapper handles interaction details and from L3 takes the canonical form `a,b,c,... = wrapper(x,y,...)`. This approach works very well, but requires a wrapper for every external function. The second approach is experimental; there is support for directly calling external programs while automatically collecting their output for further use from L3. This automatic collection includes printed output and the names of generated files, but unless these external programs have very strict rules for file naming, further use of this information is difficult.

The worksheet and special purpose interfaces

The L3 worksheet is a generic nested list manipulator and item selector and allows the user to manipulate data in any way (s)he sees fit. The ability to handle nested lists of data covers many cases of interactive data handling, and it subsumes file-manager functionality. For specialized applications, custom viewers for non-recursive data can be easily added to the system; the image display shown previously is one example. While there is no provision for constructing event-based L3 scripts, programmers can easily provide additional functionality through Python. The l3 worksheet itself is uses an event-based interface based on an object graphics canvas, as first introduced by ezd [16] and Tk [1], and is readily extended for special uses via Python or C programming. The ability of the canvas to embed other widgets such as 2D graphs or spreadsheet-style display tables can be used to construct custom applications. The advantage over constructing such

applications from scratch is the availabiliy of all of L3 (hence Python) to the application's *user*, through the graphical interface. Through judicious use of the outlining facilities, such an interface can contain all the detailed adjustments one expects in a script, while initially exposing only a minimal subset of parameters that require adjustment. The user can then be exposed to further details as necessary for his work. In analogy to programming, one can embed an interface in a worksheet, just as one can embed an application API in a language.

The structures formed through interaction

Looking at the previous example from a data structure view illustrates the source of the interaction's complexity. The original for loop section E of figure 6 produces only a *sequence* of independent `(float, float, string, string, string)` tuples. User examination of the leading `float` values requires a *list* of these tuples. Following this, examination of the file referenced by the first `string` requires one indirection to the file (`_K-8-135584/average`, ③ in figure 6), followed by display of an `(int, image) list` (❹ in figure 7).

The full session, regardless of how it is handled, is therefore a selective inspection of parts of a `(float, float, string, string, string) list` where every `string` is a reference to a file holding a `(int, image) list`. Because the program was not written with these structures in mind, they are formed through user interaction, not through any programs. Thus any assistance must come from the envionment the user interacts with.

The shell session is a manual traversal of an implicit, unformed data structure and therefore provides no support; a worksheet session is the same. Data flow environments and L3 go further and structure all data; the L3 session is a semi-automated traversal of an automatically formed, internal data structure. Unlike scripting and programming languages, data flow environments and L3 impart structure to unplanned and unplannable user activities.

The possible workflows in L3

The interaction convenience of data flow environments and L3 require restrictions on what users can do in order to provide assistance with the data structures arising from interaction. In L3, the user is restricted to tail calls as repetition mechanism – there are no arbitrary goto-like control structures.

In unconstrained control flow, the data structure is arbitrary. For this case, no general statements can be made about the structure of data produced by a user. While this allows the user to do anything at all, environments based on this approach cannot provide data structuring support.

In pure data flow the data has directed acyclic graph (DAG) structure, explicitly formed through user action. DAGs have a natural graphical representation that is fully general; no special graph drawing algorithms are required for "different" DAGs. While the overall

data structure is precisely determined at any time and environments can provide support, the user is restricted to trivial control flow. Incorporating loops is possible, but far from natural [17].

L3 uses tail calls with lexical scoping, resulting in data structured as a nested DAG. Without loops or function calls, this reduces to a simple data flow DAG. A single call is a node containing its own DAG, while a loop (or a tail call) is a node representable as a stack of DAGs, where every tail call introduces a new level.

Thus, data structures formed by L3 execution have a fixed format that generalizes the structure found in pure data flow. L3 function calls** added to a script by a user, as illustrated in the sample session, are equivalent to manual addition of a clone of the current DAG in a data flow setting. L3 function calls in loops represent programmatic cloning and embedding of existing DAGs, with selective replacements of certain values (the function arguments). This latter form has no direct analogue in data flow.

CONCLUSION

This paper presented a framework for script-based user-computer interaction that shields users from tedious bookkeeping and allows them to focus on their data and scripts. The approach pairs a language designed for interaction and simple scripting with an interface to take advantage of metadata produced by the language. The system relies on a novel combination of features in the language, viz

- unique identification of every expression and computed datum
- persistence of programs and associated data
- lexical scoping and tail calls
- programmability via conditionals and functions
- need-based execution of programs

to provide the user-visible facilities of implicit hierarchial data organization, direct access to data from scripts (and vice versus), and script evolution.

The implementation of L3 described in this paper is freely available under an open-source license at `http://l3lang.sourceforge.net`.

ACKNOWLEDGEMENTS

This research was supported by the NIH/NIGMS under grant number P01GM064692, and was performed under DOE contract No. DE-AC02-05CH11231.

**A call to a builtin function – Python or C – is considered atomic in L3 and makes no addition to the data graph.

REFERENCES

1. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.
2. R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua — an extensible extension language," *Software—Practice and Experience* **26**(6), pp. 635–652, 1996.
3. G. van Rossum, "Python reference manual," Report CS-R9525, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, Apr. 1995.
4. C. B. Moler, "Matlab — an interactive matrix laboratory," Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.
5. J. W. Eaton, "Octave: A high-level interactive language for numerical computations," Feb. 1992. Originally developed at the University of Texas in Austin, and later contributed to the GNU Project of the Free Software Foundation. Octave is Matlab-like, but is not a full reimplementation.
6. R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics* **5**(3), pp. 299–314, 1996.
7. S. I. Feldman, "Make-a program for maintaining computer programs," *Software - Practice and Experience* **9**(4), pp. 255–65, 1979.
8. SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI).
9. M. Sanner, D. Stoffler, and A. Olson, "Viper, a visual programming environment for python," in *Proceedings of thre 10th International Python conference*, pp. 103–115, February 4-7 2002.
10. T. R. G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs," in *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992.
11. P. D. Adams, R. W. Grosse-Kunstleve, L.-W. Hung, T. R. Ioerger, A. J. Mccoy, N. W. Moriarty, R. J. Read, J. C. Sacchettini, N. K. Sauter, and T. C. Terwilliger, "Phenix: building new software for automated crystallographic structure determination," *Acta Crystallographica Section D* **58**, pp. 1948–1954, 2002.
12. S. G. Parker, D. M. Weinstein, and C. R. Johnson, "The SCIRun computational steering software system," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997.
13. C. Pautasso and G. Alonso, "The jopera visual composition language," *J. Vis. Lang. Comput.* **16**(1-2), pp. 119–152, 2005.
14. P. Penczek, M. Radermacher, and J. Frank, "Three-dimensional reconstruction of single particles embedded in ice," *Ultramicroscopy* **40**, pp. 33–53, Jan 1992.
15. *GTK+ Reference Manual*. http://www.gtk.org.
16. J. F. Bartlett, "Don't fidget with widgets, draw," tech. rep., DIGITAL Western Research Laboratory, 1991.
17. M. Mosconi and M. Porta, "Iteration constructs in data-flow visual programming languages," *Computer Languages* **26**, pp. 67–104, July 2000.